

**Introduction**:

In this section we will learn about,

* Class-based views
* Generic views
* Viewsets
* Routers
* Searching, filtering and pagination

**Class-based views**:

All the views we have created so far are function based views, but Django REST framework also supports *class based views*, which *makes our code cleaner and more concise plus they provide a lot of reuse opportunities*.

We will convert this view function to a class based view,

@api\_view(["GET", "POST"])

def product\_list(request):

    if request.method == "GET":

        queryset = Product.objects.select\_related("collection").all()

        serializer = ProductSerializer(

            queryset, many=True, context={"request": request}

        )

        return Response(serializer.data)

    elif request.method == "POST":

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data, status=status.HTTP\_201\_CREATED)

🡪 First import *APIView* class

from rest\_framework.views import APIView

This is the base class for all class based views.

🡪 Now define a class called *ProductList* and have this class inherit from APIView class.

class ProductList(APIView):

Notice that for class name we are using python naming convention for classes.

🡪 Now in this class we will define two methods, a *get* method for handling get requests and a *post* method for handling post requests.

class ProductList(APIView):

    def get(self, request):

        queryset = Product.objects.select\_related("collection").all()

        serializer = ProductSerializer(

            queryset, many=True, context={"request": request}

        )

        return Response(serializer.data)

    def post(self, request):

        serializer = ProductSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data, status=status.HTTP\_201\_CREATED)

So *incoming request gets automatically dispatched to one of these methods, depending on the request type*.

First benefit of class based view is that we do not have those if elif statements.

🡪 Now in order to use this class based view, we need to go to our urls module.

urlpatterns = [

    path("products/", views.ProductList.as\_view()),

Add our *ProductList* class here with *as\_view()* method.

When we call *as\_view()* method, *it will convert this class into a regular function based view*. So ideally there is a function under the hood that gets called *but while writing our code, we use a class and that means we get access to all object oriented programming features*.

Now do the same changes for *product\_detail* view function also.

class ProductDetail(APIView):

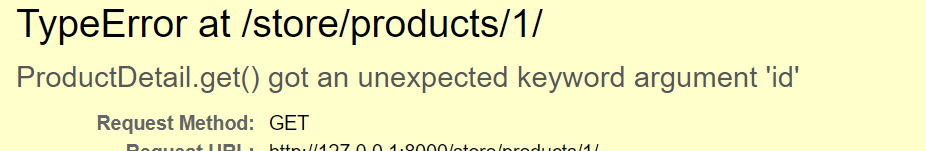
    def get(self, request):

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

But we get an error…



This is because we are passing id as parameter in the URL pattern but not defined it inside our class based View.

path("products/<int:**id**>/", views.ProductDetail.as\_view()),

After adding it in our ProductDetail,

class ProductDetail(APIView): 🡪 our class based View

    def get(self, request, id):

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

    def put(self, request, id):

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product, data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data)

    def delete(self, request, id):

        product = get\_object\_or\_404(Product, pk=id)

        if product.orderitems.count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

And our end point is working up to this point.



**Mixins**:

So we know that class based views provide a lot of reuse opportunities. Look at the get method of *ProductList* view.

class ProductList(APIView):

    def get(self, request):

        queryset = Product.objects.select\_related("collection").all()

        serializer = ProductSerializer(

            queryset, many=True, context={"request": request}

        )

        return Response(serializer.data)

There are *three common things* we are doing here…

🡪 Creating a queryset.

🡪 Creating a serializer and provide it with that queryset.

🡪 Returning a response with a serialized data.

If you notice we have the exact same pattern for our *collection\_list*.

def collection\_list(request):

    if request.method == "GET":

        queryset = Collection.objects.annotate(products\_count=Count("product")).all()

        serializer = CollectionSerializer(queryset, many=True)

        return Response(serializer.data)

Only *two differences* between these two implementations are:

🡪 How we are creating this queryset.

🡪 And the Serializer we are using.

We also see some *common pattern while creating a resource* (*POST*)

elif request.method == "POST":

serializer = CollectionSerializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data, status=status.HTTP\_201\_CREATED)

🡪 First we create a serializer to deserialize the data.

🡪 Validate the incoming data.

🡪 Save the data.

🡪 Return the response with a serialized data.

This is where *mixins* come into the picture. A *mixin* is *a class that encapsulate some pattern of code* like this.

from rest\_framework.mixins import ListModelMixin, CreateModelMixin

Look at implementation of ListModelMixin,

class ListModelMixin:

    """

    List a queryset.

    """

    def list(self, request, \*args, \*\*kwargs):

        queryset = self.filter\_queryset(self.get\_queryset())

        page = self.paginate\_queryset(queryset)

        if page is not None:

            serializer = self.get\_serializer(page, many=True)

            return self.get\_paginated\_response(serializer.data)

        serializer = self.get\_serializer(queryset, many=True)

        return Response(serializer.data)

Here we can see a class with a single method called *list* and we have a logic for listing a bunch of models.

First we are creating a queryset, then giving that queryset to a serializer and finally returning that serializer (*in between there is filtering and pagination as well, but we will look at that later*). This is very similar to pattern that we have in our code.

Now we will look at CreateModelMixin,

class CreateModelMixin:

    """

    Create a model instance.

    """

    def create(self, request, \*args, \*\*kwargs):

        serializer = self.get\_serializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        self.perform\_create(serializer)

        headers = self.get\_success\_headers(serializer.data)

        return Response(serializer.data, status=status.HTTP\_201\_CREATED, headers=headers)

    def perform\_create(self, serializer):

        serializer.save()

    def get\_success\_headers(self, data):

        try:

            return {'Location': str(data[api\_settings.URL\_FIELD\_NAME])}

        except (TypeError, KeyError):

            return {}

In this class we have a create method that *encapsulates the logic for creating a resource*.

First we create a serializer then we validate the incoming data, next we have a call to *perform\_create* method which is another method in this class (*where we call serializer.save()*).

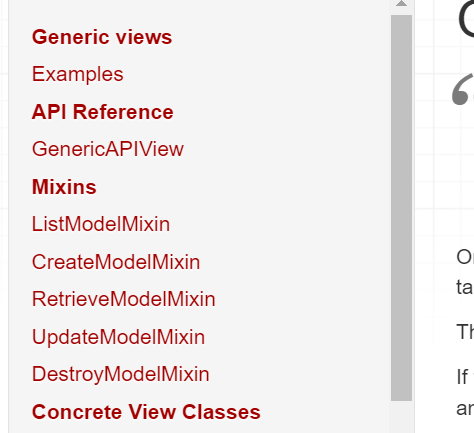
Next we include *success\_headers* to include in the response and finally we return the response with a serialized data and status code of 201.

Again very similar to how we implemented the logic for creating a product or a collection.

So in Django rest framework we have various mixins for performing different kind of operations on a resource.

Under generic views on official django rest framework page,

<https://www.django-rest-framework.org/api-guide/generic-views/>

🡨Look at all available mixins.

For example,

*RetriveModelMixin* 🡪 Helps in getting a single instance of a model.

Get to know about these mixins before jumping to next chapter.

**Generic Views**:

We learned about mixins, now most of the times we are not going to use these mixins directly.

Instead we are going to use *concrete classes that combine one or more mixins*, we call these classes *Generic Views*.

For example we have a class called ***ListCreateAPIView*** that combine two mixins *ListModelMixin* and *CreateModelMixin*.

First import it,

from rest\_framework.generics import ListCreateAPIView

And let us look at its implementation,

class ListCreateAPIView(mixins.ListModelMixin,

                        mixins.CreateModelMixin,

                        GenericAPIView):

    """

    Concrete view for listing a queryset or creating a model instance.

    """

    def get(self, request, \*args, \*\*kwargs):

        return self.list(request, \*args, \*\*kwargs)

    def post(self, request, \*args, \*\*kwargs):

        return self.create(request, \*args, \*\*kwargs)

Look, this class has multiple parents *ListModelMixin*, *CraeateModelMixin* and *GenericAPIView* (*it is the base class for all generic views*).

class GenericAPIView(views.APIView):

    """

    Base class for all other generic views.

    """

    # You'll need to either set these attributes,

    # or override `get\_queryset()`/`get\_serializer\_class()`.

    # If you are overriding a view method, it is important that you call

    # `get\_queryset()` instead of accessing the `queryset` property directly,

    # as `queryset` will get evaluated only once, and those results are cached

    # for all subsequent requests.

    queryset = None

    serializer\_class = None

    # If you want to use object lookups other than pk, set 'lookup\_field'.

    # For more complex lookup requirements override `get\_object()`.

    lookup\_field = 'pk'

    lookup\_url\_kwarg = None

    # The filter backend classes to use for queryset filtering

    filter\_backends = api\_settings.DEFAULT\_FILTER\_BACKENDS

    # The style to use for queryset pagination.

    pagination\_class = api\_settings.DEFAULT\_PAGINATION\_CLASS

This GenericAPIView class provides a bunch of methods that we are going to override in our custom views. For example *get\_queryset* (*for creating a queryset object*) and *get\_serializer\_class* (*for specifying the type of serializer we want to use in our view*).

Remember *the only difference we had in our ProductList and CollectionList views were in the queryset and serializer classes*.

Back to our generic view which is ListCreateAPIView, which provides us two methods, a *get* and *post*.

class ListCreateAPIView(mixins.ListModelMixin,

                        mixins.CreateModelMixin,

                        GenericAPIView):

    """

    Concrete view for listing a queryset or creating a model instance.

    """

    def get(self, request, \*args, \*\*kwargs):

        return self.list(request, \*args, \*\*kwargs)

    def post(self, request, \*args, \*\*kwargs):

        return self.create(request, \*args, \*\*kwargs)

A get method simply delegates to the *list* method of the current object (*list method inherited from ListModelMixin*).

class ListModelMixin:

    """

    List a queryset.

    """

    def list(self, request, \*args, \*\*kwargs): 🡨 inherited list from here...

        queryset = self.filter\_queryset(self.get\_queryset())

Similarly, post method simply delegates to the *create* method of the current object (*inherited from CreateModelMixin*)

class CreateModelMixin:

    """

    Create a model instance.

    """

    def create(self, request, \*args, \*\*kwargs):

        serializer = self.get\_serializer(data=request.data)

        serializer.is\_valid(raise\_exception=True)

        self.perform\_create(serializer)

        headers = self.get\_success\_headers(serializer.data)

        return Response(serializer.data, status=status.HTTP\_201\_CREATED, headers=headers)

So a ***Generic View*** *is a concrete class that combines one or more mixins and provides handler methods like get, post, put and delete*.

We have another generic view called *ListAPIView* that only has the listing functionality. So if we use this view we can only list our resources but cannot create them.

Now let us implement ListCreateAPIVIew in our ProductList view.

class ProductList(ListCreateAPIView):

Now we need to override two methods, get\_queryset and get\_serializer\_class,

class ProductList(ListCreateAPIView):

    def get\_queryset(self):

        return Product.objects.select\_related("collection").all()

    def get\_serializer\_class(self):

        return ProductSerializer 🡪 just the class not the object!

If you noticed earlier we had a context object inside our ProductSerializer,

        serializer = ProductSerializer(

            queryset, many=True, context={"request": request}

        )

So to use we have a method defined in GenericAPIView class called get\_serializer\_context, which we can override by returning a dictionary

    def get\_serializer\_context(self):

        return {"request": self.request}

Now our ProductList view looks much cleaner and concise then what we previously had,

class ProductList(ListCreateAPIView):

    def get\_queryset(self):

        return Product.objects.select\_related("collection").all()

    def get\_serializer\_class(self):

        return ProductSerializer

    def get\_serializer\_context(self):

        return {"request": self.request}

We can make this code even shorter, if we do not have any condition to define inside our get\_queryset or get\_serializer\_class methods (*like applying some other queryset or serializer based on user or their authorization*). Then we can directly use *queryset* and *serializer\_class* special attributes defined in our GenericAPIView base class.

class ProductList(ListCreateAPIView):

    queryset = Product.objects.select\_related("collection").all()

    serializer\_class = ProductSerializer

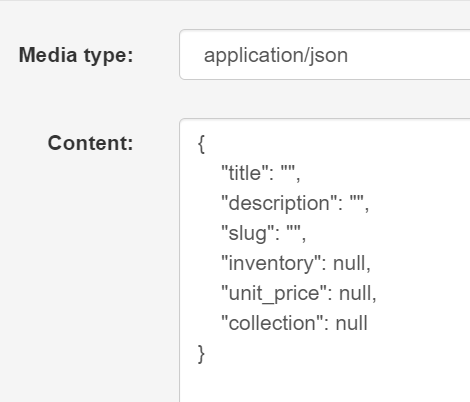
    def get\_serializer\_context(self): 🡪 no special attribute for context

        return {"request": self.request}

Check in the browsable API, we have the exact same functionality like before. But there is something cool in POST section of page.

<HTML form

And if you switch to raw data tab,

🡨 You can see a sample product object that we can send to the server.

Previously before we started GeneriAPIView, we did not have the sample but only an empty content box. So this is another benefit of generic views.

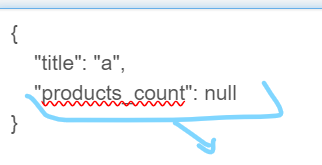
Similarly for CollectionList,

class CollectionList(ListCreateAPIView):

    queryset = Collection.objects.annotate(products\_count=Count("product")).all()

    serializer\_class = CollectionSerializer

But there is an issue in POST request,

, we do not want to specify products\_count manually here. So if we remove this field from request body we get 400 bad request.

So in CollectionSerializer we will make this field read only by passing a keyword argument read\_only and set it to True.

class CollectionSerializer(serializers.ModelSerializer):

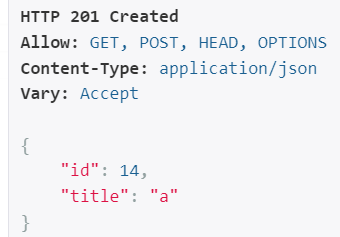
    class Meta:

        model = Collection

        fields = ["id", "title", "products\_count"]

    products\_count = serializers.IntegerField(read\_only=True)

With this we can create new collection,



**Customizing Generic Views**:

There are situations where a generic view may not quite work for us. So let us see how we can customize it.

For now we have ProductDetail view which provides three operations get, put and delete.

class ProductDetail(APIView):

    def get(self, request, id):

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product)

        return Response(serializer.data)

    def put(self, request, id):

        product = get\_object\_or\_404(Product, pk=id)

        serializer = ProductSerializer(product, data=request.data)

        serializer.is\_valid(raise\_exception=True)

        serializer.save()

        return Response(serializer.data)

    def delete(self, request, id):

        product = get\_object\_or\_404(Product, pk=id)

        if product.orderitems.count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

Now we have a generic view that provides all these operations, so to use it here we need to change base class of this view to *RetrieveUpdateDestroyAPIView*.

class ProductDetail(RetrieveUpdateDestroyAPIView):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

With these two attributes, we can remove get and put method since this pattern is already implemented in our base class.

But delete method has some logic (*if condition*) that is specific to our application. So we need to override the delete method that we have inherited from *RetrieveUpdateDestroyAPIView*. Let us have a look at this class.

class RetrieveUpdateDestroyAPIView(mixins.RetrieveModelMixin,

                                   mixins.UpdateModelMixin,

                                   mixins.DestroyModelMixin,

                                   GenericAPIView):

    """

    Concrete view for retrieving, updating or deleting a model instance.

    """

    def get(self, request, \*args, \*\*kwargs):

        return self.retrieve(request, \*args, \*\*kwargs)

    def put(self, request, \*args, \*\*kwargs):

        return self.update(request, \*args, \*\*kwargs)

    def patch(self, request, \*args, \*\*kwargs):

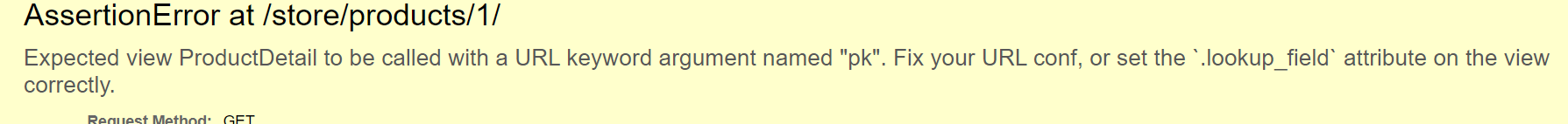
        return self.partial\_update(request, \*args, \*\*kwargs)

    def delete(self, request, \*args, \*\*kwargs):

        return self.destroy(request, \*args, \*\*kwargs)

We are using get, put, patch methods as it is. But we are overriding delete method and replacing its implementation with our specific implementation.

We get an exception,



Back to our URL’s module,

path("products/<int:id>/", views.ProductDetail.as\_view()),

Our generic view expect this URL keyword argument ‘*pk*’ instead of *id*. So change it to *pk*.

Also change it inside *ProductDetail* view as well,

class ProductDetail(RetrieveUpdateDestroyAPIView):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    def delete(self, request, pk): 🡪 change here

        product = get\_object\_or\_404(Product, pk=pk) 🡪 and here

        if product.orderitems.count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

Note: If you have a very strong reason to call your parameter to ‘*id*’ then you can set the lookup\_field parameter inside view function to id.

class ProductDetail(RetrieveUpdateDestroyAPIView):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    lookup\_field = 'id'

But it is recommended to stick with Django REST framework conventions ☺

Same implementation for CollectionDetail view,

class CollectionDetail(RetrieveUpdateDestroyAPIView):

    queryset = Collection.objects.all()

    serializer\_class = CollectionSerializer

    def delete(self, request, pk):

        collection = get\_object\_or\_404(Collection, pk=pk)

        if collection.product\_set.count() > 0:

            return Response(

                {

                    "error": "This collection cannot be deleted because it is associated with a product"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        collection.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

**ViewSets**:

So currently we have two views for managing our product. We have the *ProductList* for managing and creating products and we also have *ProductDetail* for getting, updating and deleting products.

class ProductList(ListCreateAPIView):

    queryset = Product.objects.select\_related("collection").all()

🡪 remove select\_related from here...

    serializer\_class = ProductSerializer

    def get\_serializer\_context(self):

        return {"request": self.request}

class ProductDetail(RetrieveUpdateDestroyAPIView):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    def delete(self, request, pk):

        product = get\_object\_or\_404(Product, pk=pk)

        if product.orderitems.count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

We can see some duplication (*select\_related field is not necessary to eager loading collections, so both querysets will be same*) queryset and serializer\_class are same in both.

This is where we use *viewset*, with which *we can combine the logic for multiple related views inside a single class*. That is why it’s called a viewset (*a set of related views*).

First import ModelViewSet class.

from rest\_framework.viewsets import ModelViewSet

Inside this class,

class ModelViewSet(mixins.CreateModelMixin,

                   mixins.RetrieveModelMixin,

                   mixins.UpdateModelMixin,

                   mixins.DestroyModelMixin,

                   mixins.ListModelMixin,

                   GenericViewSet):

    """

    A viewset that provides default `create()`, `retrieve()`, `update()`,

    `partial\_update()`, `destroy()` and `list()` actions.

    """

    pass

This class has multiple base classes. Mixins we are familiar with but let us look at *GenericViewSet* class.

class GenericViewSet(ViewSetMixin, generics.GenericAPIView):

    """

    The GenericViewSet class does not provide any actions by default,

    but does include the base set of generic view behavior, such as

    the `get\_object` and `get\_queryset` methods.

    """

    pass

As we can see, everything we have learned about generic views also exist in ***genericAPIViews***. So here we have those attributes like *queryset*, *serializer\_class* etc…

Now we will combine logic for *ProductList* and *ProductDetail* view using a viewset.

class ProductViewSet(ModelViewSet):

Note the naming convention of the view set, name of the model followed by ViewSet.

We move code from our views inside this viewset.

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    def get\_serializer\_context(self):

        return {"request": self.request}

    def delete(self, request, pk):

        product = get\_object\_or\_404(Product, pk=pk)

        if product.orderitems.count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

Now we have a single class for implementing the products endpoint. Using this single class, we can list our products, create, update or delete them.

This is the benefit of using view set.

But our application is broken now, since in our urls module, we are still referencing old views which are gone now.

    path("products/", views.ProductList.as\_view()),

    path("products/<int:pk>/", views.ProductDetail.as\_view()),

Next we will learn about routers and we will see how we can use router to create the route for a view set.

But first, let us create a viewset for collections as well,

class CollectionViewSet(ModelViewSet):

    queryset = Collection.objects.annotate(products\_count=Count("product")).all()

    serializer\_class = CollectionSerializer

    def delete(self, request, pk):

        collection = get\_object\_or\_404(Collection, pk=pk)

        if collection.product\_set.count() > 0:

            return Response(

                {

                    "error": "This collection cannot be deleted because it is associated with a product"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        collection.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

Note: If our viewset inherit from ModelViewSet, we can perform all kinds of operations on a resource. We can list that resource, create it, and update it and so on.

But What if we do not want to have write operations like, we do not want to be able to create / update / delete it (*only able to list and retrieve*). For this we have another class in our viewset module called *ReadOnlyModelViewSet*.

from rest\_framework.viewsets import ModelViewSet, ReadOnlyModelViewSet

With this we can only perform read operations like listing all collections or retrieving a single collection.

**Routers**:

When we use viewsets, we are not going to explicitly register these URL patterns.

urlpatterns = [

    path("products/", views.ProductList.as\_view()),

    path("products/<int:pk>/", views.ProductDetail.as\_view()),

    path("collections/", views.CollectionList.as\_view()),

    path(

        "collections/<int:pk>/",

        views.CollectionDetail.as\_view(),

        name="collection-detail",

    ),

]

Instead we are going to user *routers*. So *we registers our viewset with a router and the router will take care of generating these URL patterns for us*.

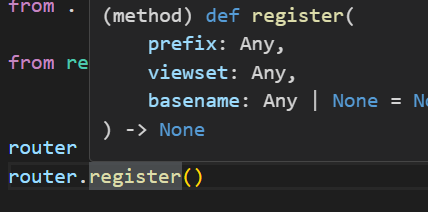
In urls.py module import *SimpleRouter* class.

from rest\_framework.routers import SimpleRouter

Next we create a SimpleRouter object.

router = SimpleRouter()

And register our viewset with this router using *router.register()*

****

We will pass two arguments here, first one is *prefix* (*name of our endpoint*) and second one is name of our viewset,

router.register('products', views.ProductViewSet)

router.register("collections", views.CollectionViewSet)

Now we can get all these urlpatterns from *router.urls*.

So before we go any further let us print it using *pprint* module.

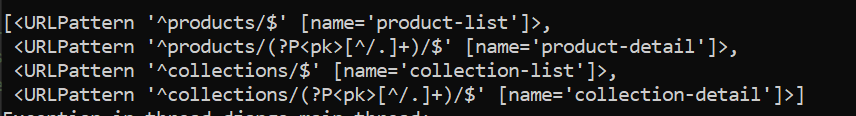
router = SimpleRouter()

router.register("products", views.ProductViewSet)

router.register("collections", views.CollectionViewSet)

pprint(router.urls)

We have an array of 4 urlpattern objects.



Look at the first urlpattern object, in the prefix, ^ represents beginning of the string and $ represents end. (*Regular expressions*).

Second name= ‘*product-list*’ is the pattern name. This name is generated based on the prefix that we specified.

Now we can specify urlpatterns to *router.urls*.

from django.urls import path

from . import views

from rest\_framework.routers import SimpleRouter

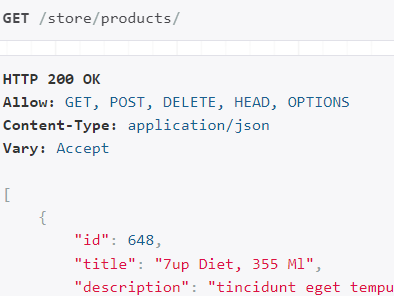
router = SimpleRouter()

router.register("products", views.ProductViewSet)

router.register("collections", views.CollectionViewSet)

urlpatterns = router.urls

With this our endpoints are now working as before.



Note: If we want to have some specific urlpatterns in our urls then we cannot set urlpatterns to router.urls. Instead we are going to create a new pattern using *path* and *include*.

urlpatterns = [

    path(" ", include(router.urls)),

    path("/reviews", views.ReviewSet.as\_view())), other url for specific purpose

]

The *DefaultRouter*:

In this routers module, we have another router called DefaultRouter. If we use this instead of SimpleRouter, we get two additional features.

router = DefaultRouter()

router.register("products", views.ProductViewSet)

router.register("collections", views.CollectionViewSet)

urlpatterns = router.urls

🡪 First one, Go to the /store (our project end point).



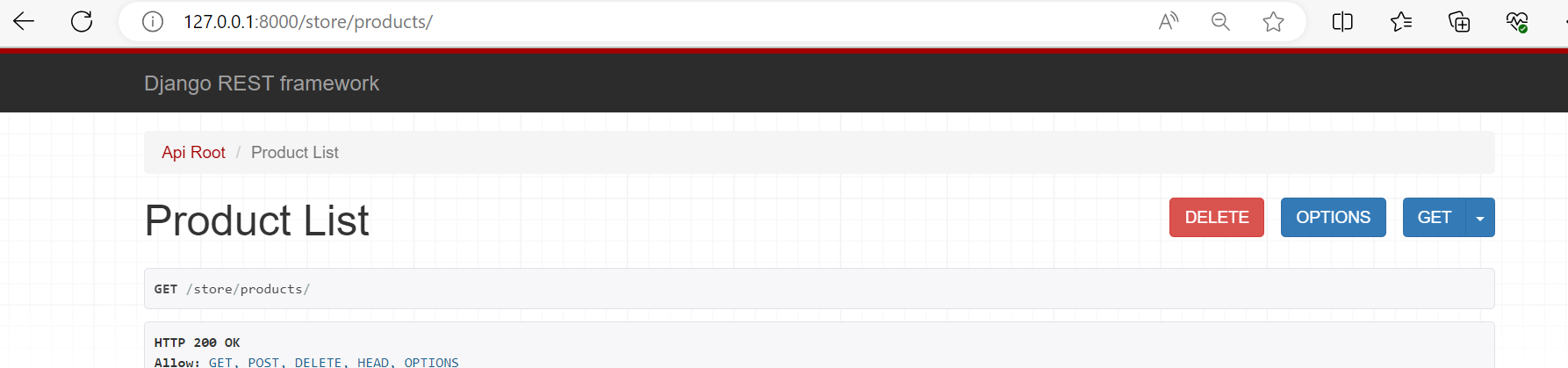
We get this page, which we call our *API route*. Here we can see various API end points available to us.

🡪 If we go to /store/products and *append .json at the end*, we see our data in JSON format.



So if a client application sends a request to this address, they get all data in JSON.

-----A little detour---- Some Issue found------



In our products endpoint we have a delete button. But this button should only appear when we are looking at a specific product (like product #1 or #2).

But here we are looking at the list of products, so it does not make any sense to have this button here.

Since the beginning it was not the case but earlier when we combined our list and detail views into ProductViewSet, *we forgot to change delete method to destroy*.

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    def get\_serializer\_context(self):

        return {"request": self.request}

    def delete(self, request, pk):

        product = get\_object\_or\_404(Product, pk=pk)

        if product.orderitems.count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        product.delete()

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

If you remember we have DestroyModelMixin in our ModelViewSet class.

class ModelViewSet(mixins.CreateModelMixin,

                   mixins.RetrieveModelMixin,

                   mixins.UpdateModelMixin,

                   mixins.DestroyModelMixin,

                   mixins.ListModelMixin,

                   GenericViewSet):

If you look at this DestroyModelMixin, we have a destroy method here.

class DestroyModelMixin:

    """

    Destroy a model instance.

    """

    def destroy(self, request, \*args, \*\*kwargs):

        instance = self.get\_object()

        self.perform\_destroy(instance)

        return Response(status=status.HTTP\_204\_NO\_CONTENT)

    def perform\_destroy(self, instance):

        instance.delete()

In destroy method, we have a call to *get\_object*() which returns the object we are looking at (*like a specific product*). It is very similar to our implementation we have in our delete function.

The only difference is the validation logic. So instead of overriding delete method, we should override *destroy* method.

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    def get\_serializer\_context(self):

        return {"request": self.request}

    def destroy(self, request, \*args, \*\*kwargs): 🡪 use kwargs for pk from URL

        if OrderItem.objects.filter(product\_id=kwargs["pk"]).count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        return super().destroy(request, \*args, \*\*kwargs)

Similarly for CollectionSet,

class CollectionViewSet(ModelViewSet):

    queryset = Collection.objects.annotate(products\_count=Count("product")).all()

    serializer\_class = CollectionSerializer

    def destroy(self, request, \*args, \*\*kwargs):

        if Product.objects.filter(collection\_id=kwargs["pk"]).count() > 0:

            return Response(

                {

                    "error": "This collection cannot be deleted because it is associated with a product"

                },

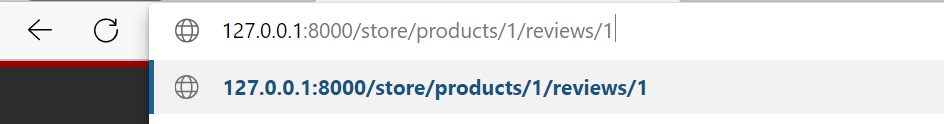
                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        return super().destroy(request, \*args, \*\*kwargs)

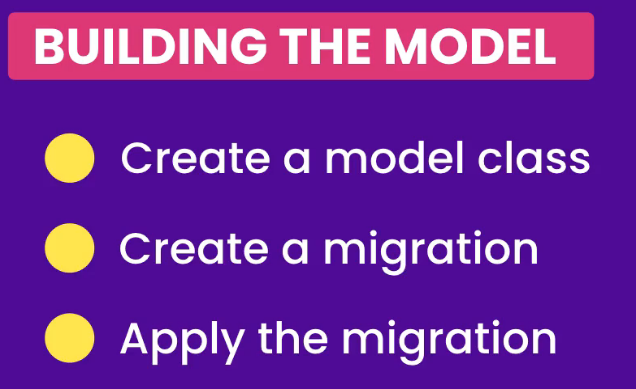
**Building the Reviews API**:

Now we are going to take our API to the next level and introduce reviews. So a given product is going to have reviews and we should be able to access an individual review like this.



So because we have *nested resources*, we need to talk about *nested routers*. But before we do that, first we need to build our model.

There are three steps for building the model,



🡪 So starting first from creating a model class called *Review* inside models.py of store app. There are four fields we need to define in here.

First one is *product* for which we are writing the review. It is a *foreign key* to product model, *on\_delete* is set to models.CASCADE (*if a product is deleted, all its reviews are deleted as well*) and Related name attribute is set to ‘*reviews’*, so that *in product class we will have an attribute called reviews*.

class Review(models.Model):

    product = models.ForeignKey(

        Product, on\_delete=models.CASCADE, related\_name="reviews"

    )

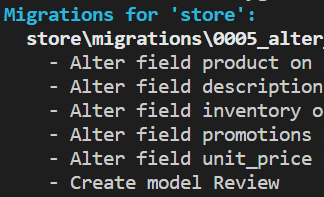
    name = models.CharField(max\_length=255) 🡪name of the reviewer

    description = models.TextField()

    date = models.DateField(auto\_now\_add=True)

🡪 Now we create a migration.

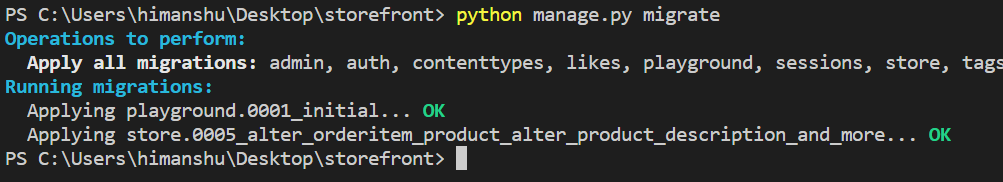
python manage.py makemigrations



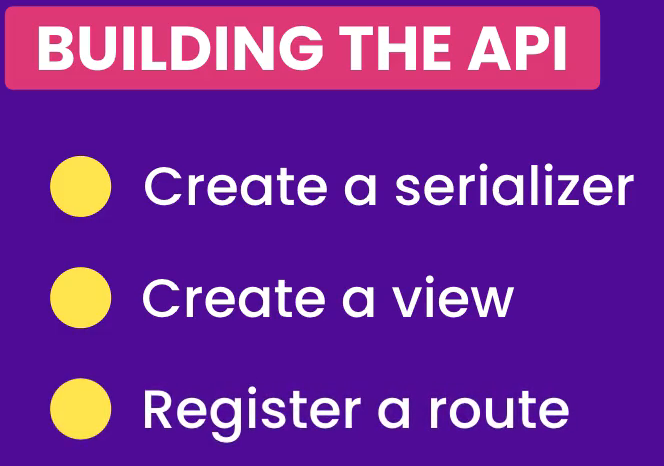
There are other changes as well as with our create model Review. (*Avoid too many changes in the migrations*)

🡪 And finally apply the migrations.

python manage.py migrate



We are done with the model, now let us work on the API. There are 3 steps we need to follow here.



🡪 Creating a *ReviewSerializer* in our serializers.py module.

class ReviewSerializer(serializers.ModelSerializer):

    class Meta:

        model = Review

        fields = ["id", "date", "name", "description"]

🡪 Let us create a viewset,

We will create a viewset and inherit it from ModelViewSet because instead of defining two separate views, one for listing reviews and others for working with an individual review, we are using a viewset that combines all operations for views inside a single class.

class ReviewViewSet(ModelViewSet):

  queryset = Review.objects.all()

  serializer\_class = ReviewSerializer

🡪 The final step is registering the routes and this is where we use nested routers.

**Nested Routers**:

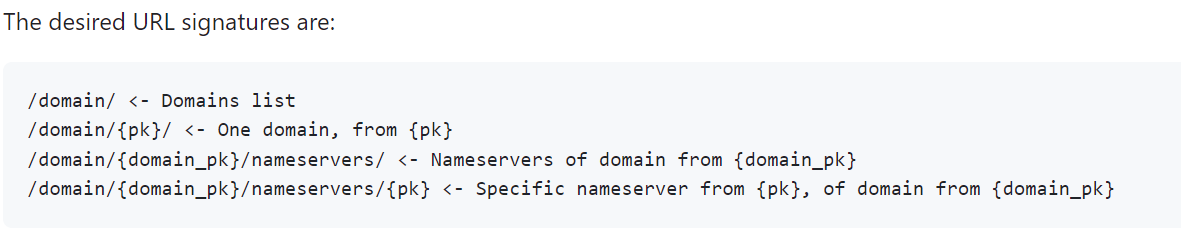
We have this project on GitHub called *drf-nested-routers*.

<https://github.com/alanjds/drf-nested-routers>

So we will see instructions on this page on how to install / use this library.

Install it using, pip install drf-nested-routers

*Under quick start*:



Here we have a *domain* and a given domain can have *nameservers*. Since we have nested resources therefore we need nested routers.

*Example*:



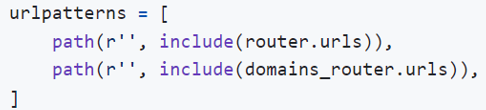
🡪 In this example, in order to implement nested routing we first create a simple router and register the parent resource *DomainViewSet*. Nothing new so far.

🡪 Now comes the new part, then we create a *NestedSimpleRouter* and give it 3 arguments. First one is *parent router*, second is *parent prefix* (‘*products’*) and third is *lookup parameter* (‘*product*’).

🡪 Then on this router we *register* the child resource. So ‘nameservers’ (*reviews*) is mapped to NameServerViewSet (*ReviewViewSet*) and *basename* is set to ‘domain-nameservers’ (*product-reviews*). This basename value will be used to generate our urlpatterns.

Note: Remember, when we use a router to register a route that router generates two URL patterns, one is called *list* and other is called *detail*. So the value of *basename = ‘domain-nameservers* will be used as a prefix for list and detail views.

So we create two routers here and then we include both the routes inside our urlpatterns array.



Now let us convert this example into products and reviews.

1. In the urls.py module, let us import routers module from rest\_framework\_nested.

from rest\_framework\_nested import routers

This routers module has a bunch of classes like DefaultRouter, SimpleRouter, NestedDefaultRouter, NestedSimpleRouter.

1. We create parent router using DefaultRouter,

router = routers.DefaultRouter()

router.register("products", views.ProductViewSet)

1. Now we create child router using NestedDefaultRouter

products\_router = routers.NestedDefaultRouter(router, "products", lookup="product")

Here in NestedDefaultRouter we pass three arguments. The parent router, parent prefix and our lookup parameter which is ‘product’ (*this means our route will have a parameter called product\_pk*).

1. Now, on this *products\_router* we are going to register our child resource.

products\_router.register('reviews', views.ReviewViewSet, basename='product-reviews')

Here we specify the prefix which is *reviews*, our viewset which is *views.ReviewViewSet* and finally basename ‘product\_reviews’ (*used as a prefix for generating the name of URL patterns*) So *our routes are going to be called* ***product-reviews-list*** *or* ***product-reviews-detail***.

1. Combine the URLs of both these routers and include them in urlpatterns object.

urlpatterns = router.urls + products\_router.urls

Note: Alternatively, if we have set urlpatterns to an array and in this array we have some explicit routes. We can use the *include* function to include the route across both these routers.

*Entire urls.py*.

from django.urls import include, path

from . import views

from rest\_framework\_nested import routers

router = routers.DefaultRouter()

router.register("products", views.ProductViewSet)

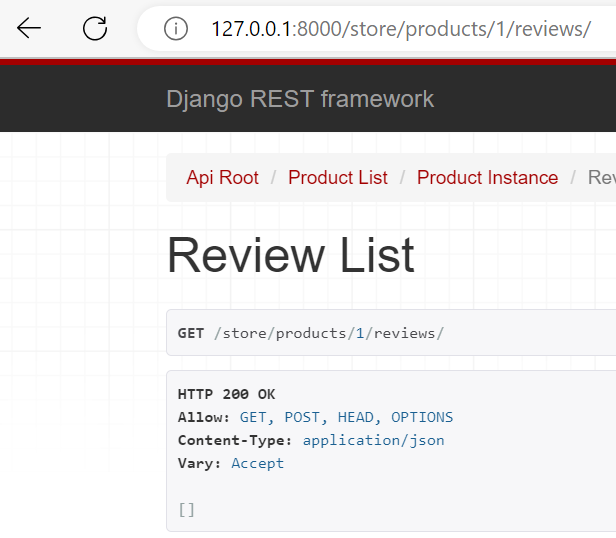
router.register("collections", views.CollectionViewSet)

products\_router = routers.NestedDefaultRouter(router, "products", lookup="product")

products\_router.register("reviews", views.ReviewViewSet, basename="product-reviews")

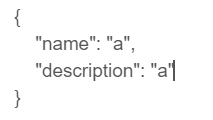
urlpatterns = router.urls + products\_router.urls

So Let us test our application up to this point,

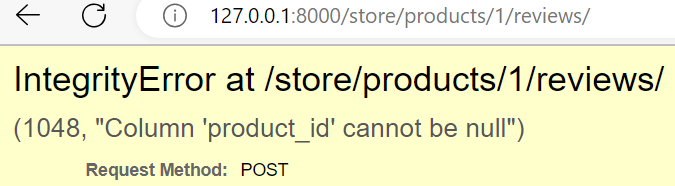


Currently we do not have any reviews. So let us post one right now.

We try to post it,



But we get an error,



It is because,

class ReviewSerializer(serializers.ModelSerializer):

    class Meta:

        model = Review

        fields = ["id", "date", "name", "description"]

In our serializer we have a save method that would either create a review or update it. Now *when creating a review it would take all these values and use them to set various fields of reviews object*. We do not have a product\_id here and that is why we are getting that exception.

This product\_id we must read it automatically from the URL, But how?

In our ViewSet,

class ReviewViewSet(ModelViewSet):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

We have access to URL parameters. So we can *read the product\_id from the URL and using a context object, we can pass it to serializer*. (*We use context object to provide additional data to a serializer*)

class ReviewViewSet(ModelViewSet):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    def get\_serializer\_context(self):

        return {"product\_id": self.kwargs["product\_pk"]}

So here we override get\_serializer\_context method and return a dictionary. In this dictionary, we are going to add a key called product\_id and set it to self.kwargs[ ‘product\_pk’].

(*This is a dictionary that contains our URL parameters*).

From here we are going to extract ‘*product\_pk*’. Remember our URL has two parameters *product\_pk* and *pk*.

So we pass this dictionary to our serializer.

Now in our serializer, we are going to override the create method for creating a review.

class ReviewSerializer(serializers.ModelSerializer):

    class Meta:

        model = Review

        fields = ["id", "date", "name", "description"]

Instead of relying on default implementation that will simply get the defined field values and set various fields in a review object. Instead we are going to provide our own implementation.

class ReviewSerializer(serializers.ModelSerializer):

    class Meta:

        model = Review

        fields = ["id", "date", "name", "description"]

    def create(self, validated\_data):

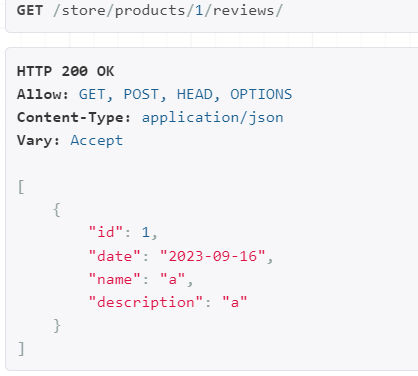
        product\_id = self.context["product\_id"]

        return Review.objects.create(product\_id=product\_id, \*\*validated\_data)

🡪 First we read product\_id from self.context[‘product\_id’] (*context is simply a dictionary and we read ‘product\_id’ property from that dictionary*)

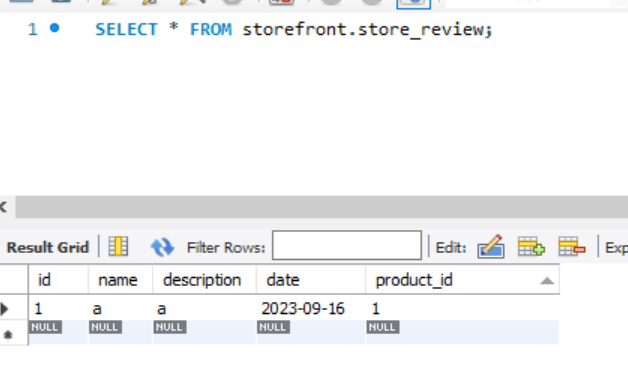
🡪 Then we unpack validated\_data dictionary and return this product object from this create method.

Next we test our implementation,



Now we can successfully post a review for a product.

In our database, we can see our review,



This review is assigned to product\_id = 1

But if we go to product/2 on our page, we still see this review,

🡨Only product\_id =1 should be able to see this review not any other product.

So to solve this problem. Go to our view,

class ReviewViewSet(ModelViewSet):

    queryset = Review.objects.all()

    serializer\_class = ReviewSerializer

    def get\_serializer\_context(self):

        return {"product\_id": self.kwargs["product\_pk"]}

Here we set our queryset to Review.objects.all(). So all reviews are returned no matter what products we are looking at.

So here we need to apply a filter based on product\_id, but for doing that we need to override get\_queryset method (*because to filter we need access to* ***self****.kwargs[] context*).

class ReviewViewSet(ModelViewSet):

    serializer\_class = ReviewSerializer

    def get\_queryset(self):

        return Review.objects.filter(product\_id=self.kwargs["product\_pk"])

    def get\_serializer\_context(self):

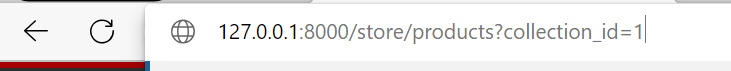
        return {"product\_id": self.kwargs["product\_pk"]}

Problem solved, we do not see any review for product#2 but if we go to product#1, we see one review.

**Filtering**:

Currently when we hit our products endpoint we see all products, but what if we want to filter them by a specific collection.

We should be able to pass a query string parameter like below,



And see only the products which belong to collection\_id = 1.

Let us see how we can implement this,

Back to ProductViewSet class

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    def get\_serializer\_context(self):

        return {"request": self.request}

    def destroy(self, request, \*args, \*\*kwargs):

        if OrderItem.objects.filter(product\_id=kwargs["pk"]).count() > 0:

            return Response(

                {

                    "error": "This product cannot be deleted because it is associated with an order item"

                },

                status=status.HTTP\_405\_METHOD\_NOT\_ALLOWED,

            )

        return super().destroy(request, \*args, \*\*kwargs)

We have set queryset to Product.objects.all(). But here we need to apply a filter. But in the previous lesson we saw that we cannot apply filter method here (*need access to self*).

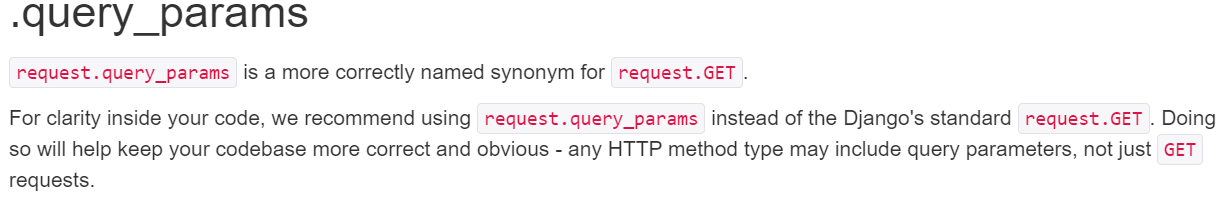
So we use get\_queryset method instead of queryset attribute.

def get\_queryset(self):

        return Product.objects.filter(collection\_id = self.request.query\_params['collection\_id'])

In the filter we set collection\_id to self.request.query\_params[‘collection\_id’]

Here **query\_params** is a dictionary which we use to read ‘collection\_id’



But what if we do not have *collection\_id* in our request, how can we handle that case?

    def get\_queryset(self):

        queryset = Product.objects.all()

        collection\_id = self.request.query\_params["collection\_id"]

        if collection\_id is not None:

            queryset = queryset.filter(collection\_id=collection\_id)

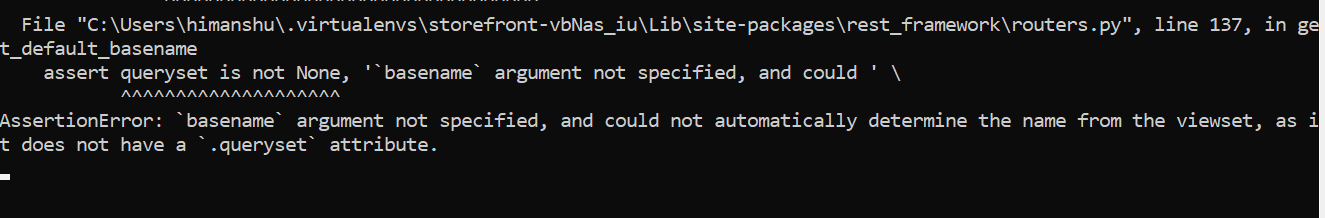
        return queryset

First we define a queryset and set it to Product.objects.all(), this is our first queryset.

Then we try to read collection\_id from a query string using query\_params.

If collection\_id is not None, that is where we apply filter and return our second queryset.

When we try to load our /store/products end point, it is not loading. In terminal we see this error,



AssertionError: `basename` argument not specified, and could not automatically determine the name from the viewset, as it does not have a `.queryset` attribute.

We removed queryset attribute from our ProductViewSet since we override it using get\_queryset method. So DRF cannot figure out the base name.



By basename I means the basename attribute that we set while registering our route (*basename is used to generate the name of our urlpatterns*). Now *by default drf uses queryset attribute to figure out basename*. Since we deleted queryset attribute and now we have a method DRF cannot figure out basename based on our custom logic.

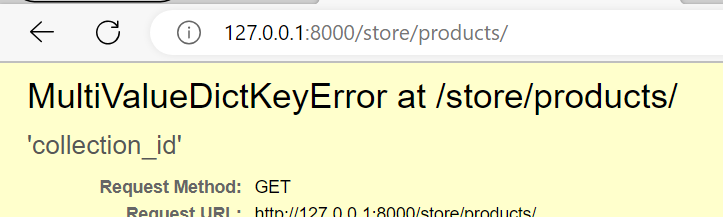
So in our URLS module,

router = routers.DefaultRouter()

router.register("products", views.ProductViewSet, basename="products")

So with this we will have two URL patterns *products-detail* and *products-list* as we know from before. It is just a prefix.

We refresh the /store/products/ end point and get a new error,



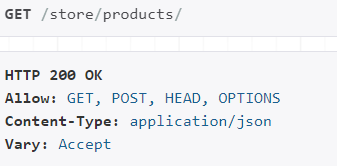
The reason we get this error, if because of this line…

collection\_id = self.request.query\_params["collection\_id"]

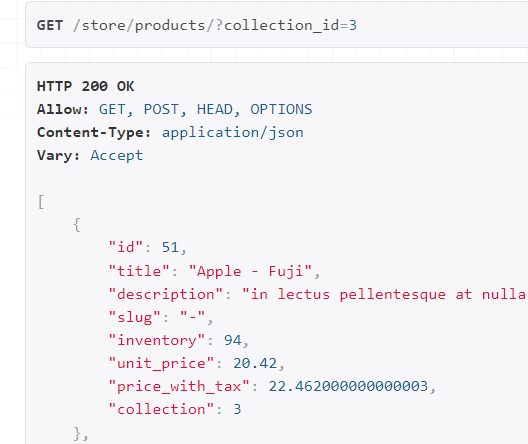
We know that query\_params is a dictionary and *we are assuming that this dictionary will always have a key called “collection\_id”*. So when we do not have this key in our dictionary we get MultiValueDickKeyError.

So to solve this issue, use *get* method which returns *none object* instead of error.

Now we get 200 OK response,



And when we filter it via collection\_id,



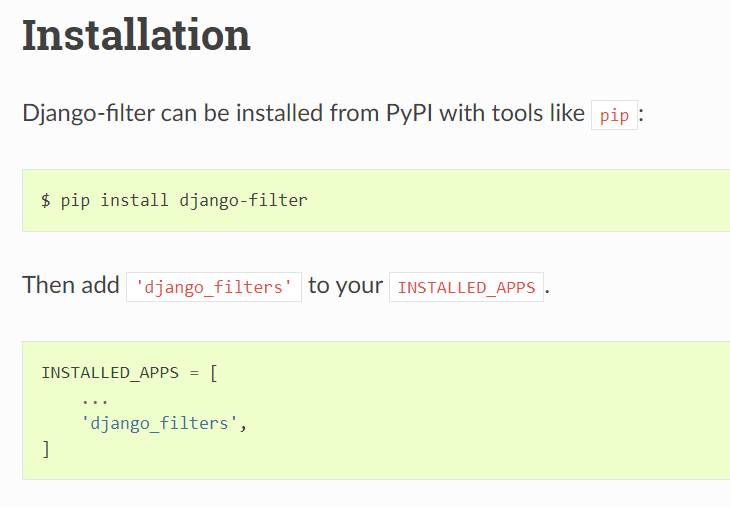
We see all the products under that collection.

**Generic Filtering**:

So we implemented basic filtering, but what if in addition to collection, we want to filter products by another field then our logic is going to get more complicated. This is where we can use *generic filtering*.

*Read the documentation*:

<https://django-filter.readthedocs.io/en/stable/>



With django-filter, we can filter any models by any fields. We do not have to hand code filtering logic like in last lecture.

Import it into our views like this,

from django\_filters.rest\_framework import DjangoFilterBackend

This *DjangoFilterBackend* gives us generic filtering.

filter\_backends = [DjangoFilterBackend]

We go to our ProductViewSet and set *filter\_backends* attribute to an array to DjangoFilterBackend. *With this backend all we have to do is specify what fields we want to use for filtering*.

filterset\_fields = ["collection\_id"]

So we set *filterset\_fields* to an array of fields.

class ProductViewSet(ModelViewSet):

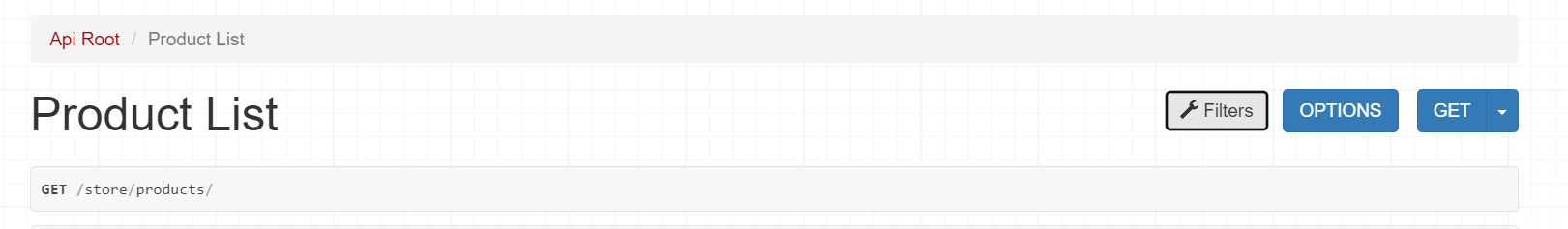
    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    filter\_backends = [DjangoFilterBackend]

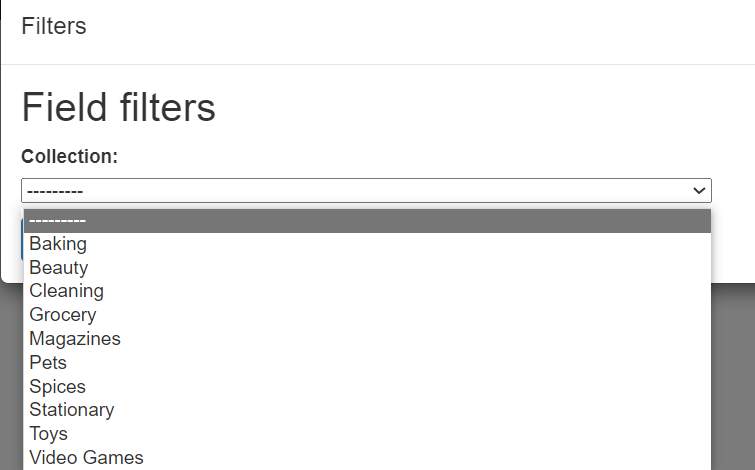
    filterset\_fields = ["collection\_id"]

And bring back our old logic…

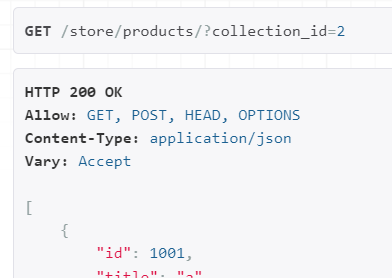


We see a *filters* icon on our browsable API page as well.

Now, we can filter on the basis of our collections,



And see the filtering in action,



Now let us take filtering to next level. What if we want to filter on the basis of *unit\_price*?

We can simply add unit\_price field in our filterset\_fields. But it would not give us a range of price which we can use to filter products. So this is where we need to implement a *custom filter*.

Note: Look at the Django filter documentation for creating custom filters.

But let us create one for unit\_price right away.

In the store app, create a file called *filters.py*.

from django\_filters.rest\_framework import FilterSet

First we import *FilterSet* class.

class ProductFilter(FilterSet):

    class Meta:

        model = Product

        fields = {}

Then create a class called *ProductFilter* which will extend *FilterSet*. And in this class we define a *Meta* class where we set *model* to Product and *fields* attribute to a dictionary (*instead of using an array we use dictionary because for each field we need to specify how the filtering should be done*).

        fields = {

            'collection\_id':['exact'],

            'unit\_price':['gt', 'lt']

        }

For collection\_id we will use **‘***exact***’** filtering (*for equality* ***==*** *comparison*). But for unit\_price we will use **‘***gt***’** and **‘**lt**’** (*greater than or less than*).

Now in our viewset we will use *filterset\_class* instead of filterset\_fields.

class ProductViewSet(ModelViewSet):

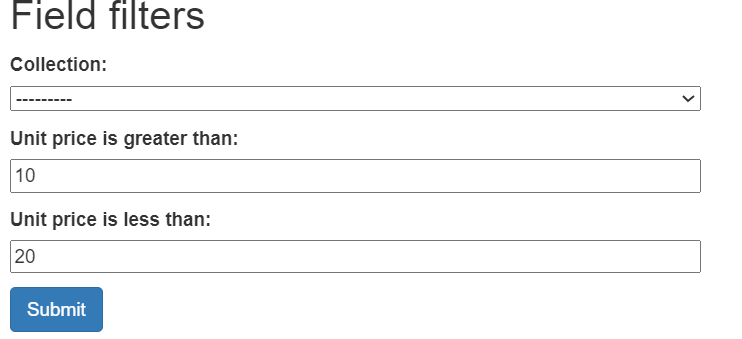
    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

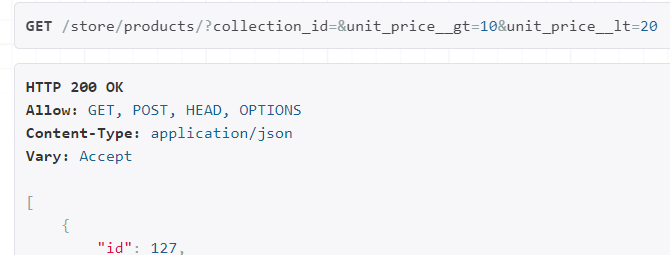
    filter\_backends = [DjangoFilterBackend]

    filterset\_class = ProductFilter

On our page,



All products with unit\_price greater than 10 and less than 20.



Look at the query parameters. You can see this library takes care of reading all these query string parameters and filtering our product by these values.

**Searching**:

What if we want to search our product by title or description? This is where we use searching. So searching is for text based fields. Let us see how we can implement this.

from rest\_framework.filters import SearchFilter

First we import *SearchFilter* class. This is another filter backend.

Back in our ProductViewSet,

filter\_backends = [DjangoFilterBackend, SearchFilter]

We add *SearchFilter* in our filter\_backends array.

search\_fields = ["title", "description"]

Set search\_fields attribute to the list of fields you want to use for searching. We can also reference fields in related classes (*like collection\_\_title*).

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

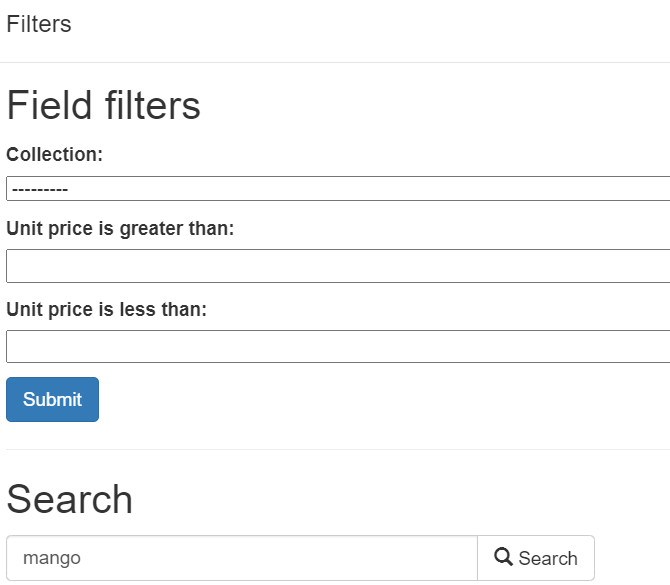
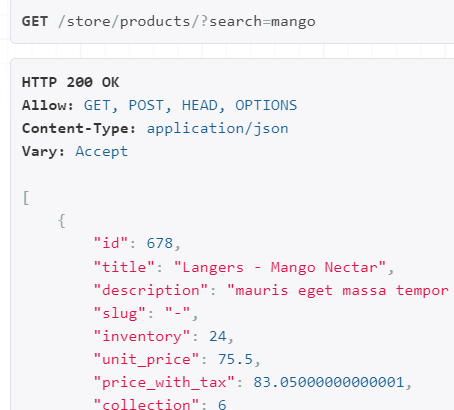
    serializer\_class = ProductSerializer

    filter\_backends = [DjangoFilterBackend, SearchFilter]

    filterset\_class = ProductFilter

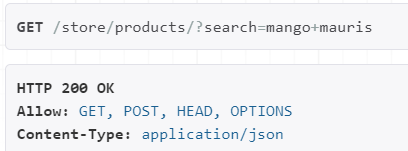
    search\_fields = ["title", "description"]

Now we refresh the page, see filters and inside there is a Search field.

By default our search is case insensitive.

We can also search by multiple expressions at once.



**Sorting**:

From the same module which is rest\_framework.filters we will import another filtering backend that is *OrderingFilter*.

from rest\_framework.filters import OrderingFilter

Then add it in list of filter\_backends.

filter\_backends = [DjangoFilterBackend, SearchFilter, OrderingFilter]

And just how we specified our search\_fields, here we can specify our *ordering\_fields* (*which we can set to list of fields we want to use for sorting*).

ordering\_fields = ["unit\_price", "last\_update"]

Our code looks like this now,

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

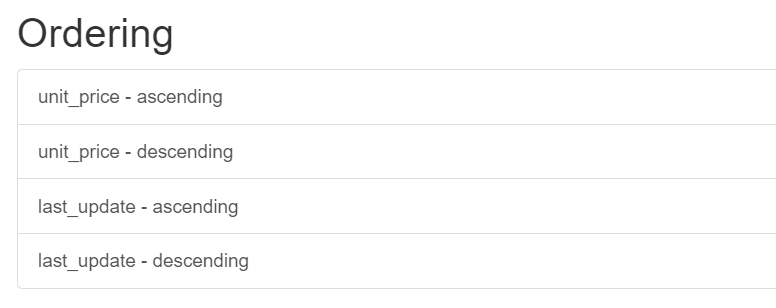
    filter\_backends = [DjangoFilterBackend, SearchFilter, OrderingFilter]

    filterset\_class = ProductFilter

    search\_fields = ["title", "description"]

    ordering\_fields = ["unit\_price", "last\_update"]

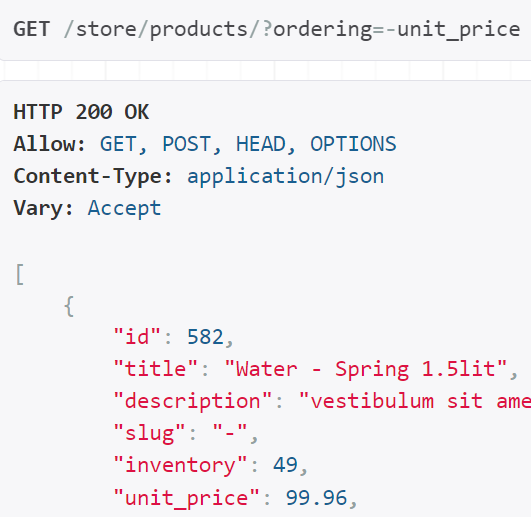
In our filters we can see various ways to sort our data.



Let us try it in ascending order of unit\_price,



In the URL we have a query string parameter of ordering. Now if you want to order by unit\_price in descending order we prefix it by a **–** sign.

🡨Now most expensive item is on top.

**Pagination**:

In our view module import *PageNumberPagination* class.

from rest\_framework.pagination import PageNumberPagination

Using this class we can paginate data using a page number (*means we can go to page#1, page#2 and so on…*).

In our viewset we will use this class.

pagination\_class = PageNumberPagination

Now we need to specify our page size. For that we need to go to our settings module.

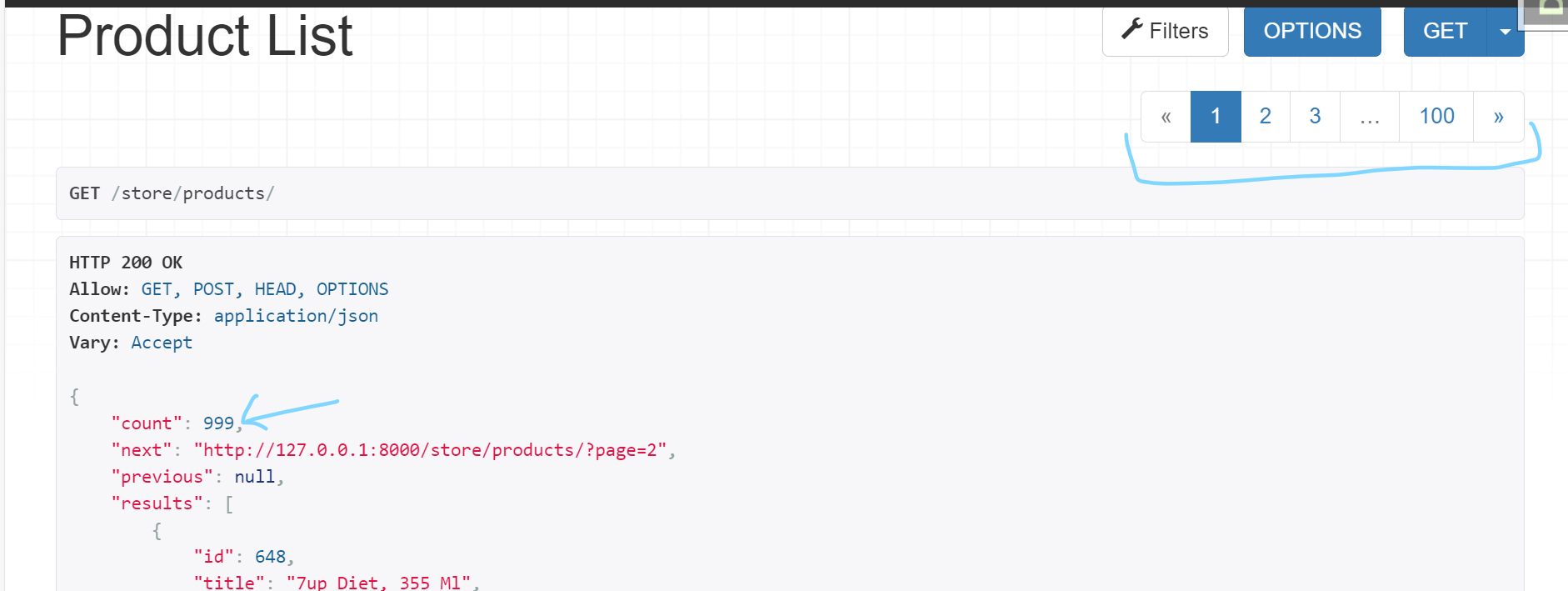
REST\_FRAMEWORK = {

    "COERCE\_DECIMAL\_TO\_STRING": False,

    'PAGE\_SIZE':10

    }

Back to our browser,



Instead of an array of products we get an object with these properties,



So we have *count*, which is total number of products in our database.

Then we have *next*, which is a link to the next page. Since we are on the first page, *previous* is null otherwise this will be a link to the previous page. And last we have *results* which is an array of products.

Now this pagination is only available for the products endpoint. So if you go to collections end point, we do not have pagination there.

*If you want to have pagination everywhere, we can set it globally in our settings module*. (*Using DEFAULT\_PAGINATION\_CLASS*)

REST\_FRAMEWORK = {

    "COERCE\_DECIMAL\_TO\_STRING": False,

    "PAGE\_SIZE": 10,

    "DEFAULT\_PAGINATION\_CLASS": "rest\_framework.pagination.PageNumberPagination",

}

We have another pagination class called *limit offset pagination*. So instead of using a page number, we use a limit and an offset value. (*Not used quite often*)

Back to our settings module, change the DEFAULT\_PAGINATION\_CLASS.

REST\_FRAMEWORK = {

    "COERCE\_DECIMAL\_TO\_STRING": False,

    "PAGE\_SIZE": 10,

    "DEFAULT\_PAGINATION\_CLASS": "rest\_framework.pagination.LimitOffsetPagination",

}

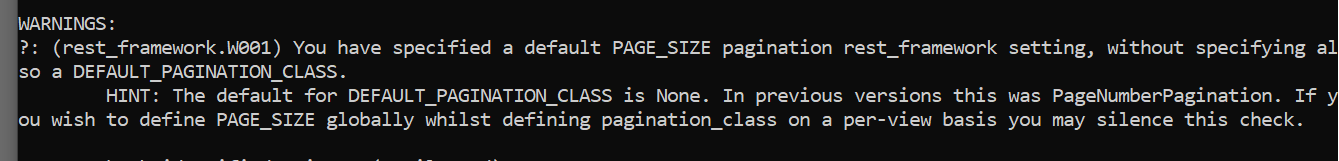
Our query string parameters have changed. Now instead of a page number we have a limit and an offset. So for the next page, we are going to take 10 products (*limit=10*) and we are going to skip 10 products (*offset=10*).



This is another way to paginate data but quite often, we use page number pagination.

We do not want to use DEFAULT pagination class everywhere, so just remove it from settings and bring back pagination\_class attribute in our ProductViewSet.

However in terminal we have a warning,



You have specified a default PAGE\_SIZE pagination rest\_framework setting, without specifying also a DEFAULT\_PAGINATION\_CLASS.

It is because in our settings module, we removed the default pagination class but we left the PAGE\_SIZE setting.

REST\_FRAMEWORK = {

    "COERCE\_DECIMAL\_TO\_STRING": False,

    "PAGE\_SIZE": 10,

}

There are two ways around this. One way is to suppress the warning (*not recommended*) by using a specific warning code and include it in rest\_framework settings.

Other approach is to use a *custom pagination class* and set the page\_size there.

🡪 Create a new file in store app called *pagination.py*.

🡪 Import *PageNumberPagination* class and create a custom class called *DefaultPagination* which inherits from it.

from rest\_framework.pagination import PageNumberPagination

class DefaultPagination(PageNumberPagination):

🡪 Here you can set page\_size attribute.

from rest\_framework.pagination import PageNumberPagination

class DefaultPagination(PageNumberPagination):

    page\_size = 10

Back to our view,

class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all()

    serializer\_class = ProductSerializer

    filter\_backends = [DjangoFilterBackend, SearchFilter, OrderingFilter]

    filterset\_class = ProductFilter

    pagination\_class = DefaultPagination 🡪 PAGINATION CLASS

Now our warning is gone.